Dynamic Programming

Dynamic Programming in a Nutshell

Dynamic Programming is an algorithmic technique for solving *optimization problems* which exhibit two properties:

1. **Optimal substructure:** the optimal solution of a problem with given input can be constructed by extending optimal solutions to some of its subproblems (or, contains within it optimal solutions to its subproblems). (*Note:* Optimal substructure property was also necessary for the greedy algorithms to work).

2. **Overlapping subproblems:** the process of constructing the optimal solution of a problem from the optimal solution of its subproblems *uses the optimal solutions of some subproblems* ***multiple times***.

Note: If there are no overlapping subproblems in an optimization problem, then a simple recursive *divide-and-conquer* approach (to be revisited later in the course) will suffice.

In essense, dynamic programming is a technique that **avoids** recomputation of the overlapping subproblems.

Dynamic programming algorithms:

- Construct and optimal solution to each subproblem encountered **exactly once**.
- The constructed optimal solution **is stored** for the duration of the algorithm execution.
- Each time an optimal solution for a subproblem is needed, the stored solution **is retrieved and used**.

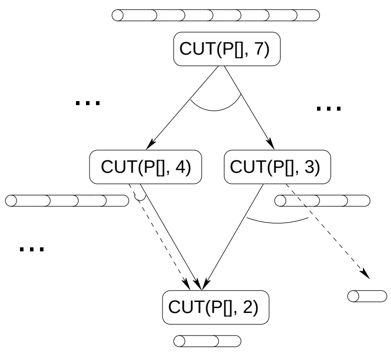

Figure 1: Overlapping subproblems for Rod Cutting problem: optimal solution for CUT(P[],2) is needed to construct optimal solutions for both CUT(P[],3) and CUT(P[],4).

Overlapping Subproblems

Overlapping subproblems. Consider an optimization problem P. We say that two subproblems P_1 and P_2 of P **overlap**, **iff** there exists a subproblem P' of P, such that optimal solutions for **both** P_1 and P_2 use the optimal solution for P'.

Example. Figure 1 shows the overlapping subproblems in the Rod Cutting problem. One of possible ways to cut a rod of size 7 is to split it into rods of sizes 3 and 4, and then construct optimal cuts for each of the two parts. Each of the two smaller rods, in turn, can have a rod of size 2 cut out of it. Therefore, optimal solutions for the Rod Cutting problem for rods of sizes 3 and 4 *both* depend on the optimal solution of the Rod Cutting problem for rods of size 2.

Why overlapping subproblems? Naïve solutions to problems with *overlapping subproblems*:

- will solve each subproblem multiple times;
- leading to increase in computational complexity of the algorithm.

Dynamic Programming algorithms solve each subproblem exactly once!

Dynamic Programming Algorithms

Two approaches:

Bottom-up: Starting with the simplest subproblem, find solutions to each subproblem in turn, until the full problem is solved.

Top-down with memoization: Break the solution of the problem into solutions of subproblems. When solving each subproblem, check if it has been solved before. *If not:* solve it and store the solution. *If yes:* use the stored solution.

Memoization: the technique of storing subproblem solutions in a (typically recursive) top-down algorithm. Involves maintaining a data structure for storing solutions, and passing this data structure around in the recursive calls.

Bottom-Up Dynamic Programming Algorithms

A bottom-up dynamic programming algorithm typically has the following features:

- **Iterative approach.** The main part of the algorithm is a collection of nested loops that iterate over the problem structure.
- **Iteration over the size of the problem.** The outer-most loop iterates over the size of optimization problem, starting with the smallest problem. Each iteration produces an optimal solution for a problem of the size given but the loop iterator variable.
- **Storage of optimal solutions.** A global data structure (typically, an array) storing optimal solutions to the problems of every size is maintained by the algorithm.

  At the end of each outer loop iteration, the computed solution is added to the data structure.
- **From small to large problems.** The optimal substructure property of the optimization problem is used on each iteration step of the algorithm to:
  - represent the optimal solution of the problem of a given size via the optimal solution of subproblems.
  - retrieve optimal solutions of the subproblems from the data structure.
  - assemble the optimal solution of the problem from the retrieved solutions.

  Note: The key difference between dynamic programming algorithms and greedy algorithms is in this step. While greedy algorithms *pursue a single solution* within the optimal substructure of problem, dynamic proramming algorithms
  - check all possible "splits" of a problem of size N into problems of smaller sizes;
  - select the best solution on each step.

  (i.e., where greedy algorithms call only one set of subproblems, dynamic programming algorithms call multiple sets of subproblems and then pick the best solution from the returned ones).

Bottom-up dynamic programming algorithm framework. Bottom-up dynamic programming algorithms are typically organized as follows:

```
ALGORITHM DPBottomUp(Data, N)  //N is the size of problem
begin
  Initialize Solutions[1..N];   //array of solutions
  for i = 1 to N do
    Splits ← { all splits of problem of size i into subproblems}
    for each s ∈ Splits do
      for each problem size j ∈ s do
        Retrieve Solutions[j];
     endfor
      Assemble TempSolution[s];  //from all Solutions[j] retrieved above
    endfor
    Solutions[i]← best solution from TempSolution[s];
  endfor
  return Solutions[N];
end
```

Example. The bottom-up solution for the Rod Cutting problem follows the organization shown above. In the case of the Rod Cutting problem, we split the problem of cutting the rod of size i, $i/2$ times. Each split is a pair of Rod Cutting problems of sizes j and $i - j$ (for every $1 \leq j \leq i$).

Top-down dynamic programming algorithms

As seen on the example of the Rod Cutting problem, a *straightforward* top-down recursive algorithm for problems with subproblem overlap is inefficient.

We can inprove top-down algorithms using **memoization**.

Memoization

Memoization is a programming technique of storing *computed results* of (typically recursive) function calls, *for the purpose of avoiding repeated computations of already established results.*

Implementing Memoization. To implement *memoization*, the algorithm needs to maintain a data structure that contains all currently computed solutions. This data structure:

- is passed to each recursive function call;
- can be modified within any of the calls;
- shall have modifications persist.

In typical programming environments this can be achieved in one of the following ways:

1. Maintaining a global variable for the solutions data structure (wherever global variables are allowed).

```
ALGORITHM MemoizedTopDown(Data, N)  //N is the size of problem
begin
  Initialize Solutions[1..N];   //memoized data structure
  return MemoizedSolution(Data, Solutions, N);
end
```

```
ALGORITHM MemoizedSolution(Data, Solutions, i)  //N is the size of problem
begin
  if Solutions[i]≠ then
    return Solutions[i]
  else
    Splits ← { all splits of problem of size i into subproblems}
    for each s ∈ Splits do
      for each problem size j ∈ s do
        call MemoizedSolution(Data,Solutions,j);
     endfor
      Assemble TempSolution[s];  //from all recursive calls
   endfor
    Solutions[i]← best solution from TempSolution[s];
   return Solutions[i];
  endif
end
```

Figure 2: Outline of a top-down dynamic programming algorithm using memoization.

2. Maintaining a class instance variable for the solutions data structure, that is accessible within the recursive function solving the optimization problem (works for Java, for example).
3. Passing the data structure to each recursive function by reference (i.e., as and in-out parameter). (whenever pass-by-reference is allowed)

Top-down dynamic programming algorithms with memoization

Using *memoization*, *straightforward* top-down recursive algorithms can be modified to behave as follows:

- **Data structure for memoization.** The algorithm maintains an appropriate data structure (typically, an array) to store solutions to optimization problems of different sizes obtained throughout the run-time of algorithm.

  This data structure is similar to the one used by the bottom-up dynamic programming algorithms. The main difference is in **how** it is used.

- **Recursive step.** On each recursive call, the function performs as follows:

  1. Checks the *memoized data structure* to see if it contains the solution to the current subproblem.
  2. If it does, returns the solution from the *memoized data structure*.

```
ALGORITHM RodCutTopDown(P[1..N], N)
begin
  Solutions[1..N];
  for i = 1 to N do      Solutions[i]← −∞;
  return RodCutMemoized(P[1..N], Solutions[1..N], N);
end
```

```
ALGORITHM RodCutMemoized(P[1..i], Solutions[1..i], i)
begin
  if Solutions[i]≥ 0 then
   return Solutions[i]
  else
   if i = 0 then       Solutions[i]← 0;
   else
    tmp ← P[i];
    for j = 1 to i do
     tmp ← max( tmp, p[j]+ RodCutMemoized(P,Solutions,i-j));
    endfor;
    Solution[i]←tmp;
   endif
  endif
  return Solutions[i];
end
```

Figure 3: Using memoization to solve the Rod Cutting problem.

3. If it does not, computes the solution to the current problem by exploring every possible split and assemblying the solution recursively, from the solutions of the subproblems.
4. Stores the newly computed solution in the *memoized data structure*. (**Note:** this is the step that was crucially missing from the *straightforward* algorithm)
5. Returns the computed solution.

Memoized Top-down algorithm pseudocode. The pseudocode for a typical top-down memoized algorithm looks as shown below. We use two functions. First function initializes the memoized data structure, and calls the solution algorithm. The solution algorithm recursively solves the problem, checking the *memoized data structure* as it proceeds as shown in Figure 2.

Top-down dynamic programming algorithm for Rod Cutting problem

The top-down dynamic programming algorithm for the Rod Cutting problem is shown in Figure 3.